Tetris game on a Field Programmable Gate Array (FPGA)

Balazs Nagy

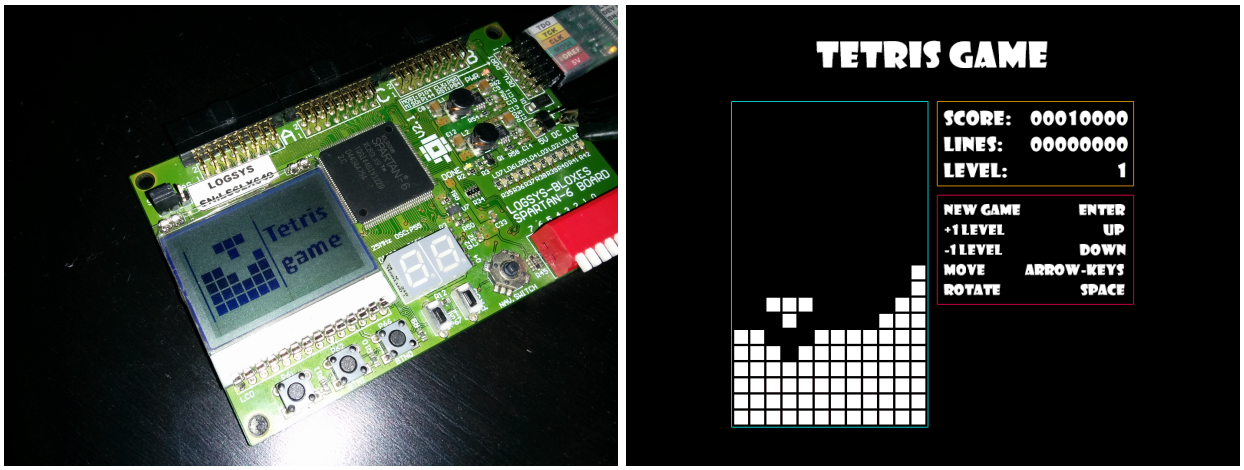
June 2015

1 Introduction

This project implements a simple Tetris game [3]: with seven different building blocks, one needs to build entire rows, such that the fully populated rows disappear, while rows that contain empty cells remain in the game. The user can manipulate the currently falling element by horizontal shifts and rotations. Blocks that already landed cannot be moved or interacted with anymore. In case the visible game space is filled, and the next falling element hits an already landed block, the game ends. The player needs to make certain, that he/she clears the rows before they fill up the available game space.

The game counts both the filled rows and the accumulated points: every row is worth 100 points. However, when multiple rows are filled and eliminated simultaneously, their worth is increased: for two consecutive lines, the game awards (100+200) points, for three lines (100+200+400), and so on. Different difficulties are also supported: when the game difficulty is increased, the speed at which the blocks are falling increases as well.

The implementation contains the original Tetris game melody [4], which is played back through a piezoelectric buzzer. It also changes its playback rate according to the difficulty selected. The game can be viewed simultaneously on the FPGA board's LCD and a regular PC monitor with a VGA interface. On the physical hardware, we only display the eliminated row count on a 7-segment display, while on the VGA output, we display both the row count and the acquired points. For control options, the user can either use the FPGA board's physical buttons or a regular PS/2 keyboard.



(a) SPI LCD (102x64 pixels, 1-bit)

(b) VGA display (800x600 pixels, 72Hz, 2-2-2-bits)

Figure 1: Visualization options of the implemented Tetris game

2 System overview

The implementation is divided into independent functional hardware blocks:

- **SPI** (Serial Peripheral Interface): low-level peripheral, responsible for the communication with the LCD.
- **Command FIFO**: buffers the commands to be sent over *SPI* to the physical LCD.
- **LCD interface**: it fills the *Command FIFO* with initialization commands and pixels to be displayed.
- **Frame buffer and updater**: it maps the current game space to the LCD's pixel space.
- **VGA interface**: it maps the current game space and state to a PC monitor.
- **Note player**: it supplies the melody of the Tetris game via a piezoelectric buzzer. The melody speeds up as the game difficulty increases.
- **PRBS** (Pseudo Random Binary Sequence) generator: this module generates pseudo-random numbers that are directly used for selecting the next falling building block.
- **Rate generator**: it manages the speed of the falling blocks and the rate of the game melody.
- **CPLD interface**: some onboard buttons and a 7-segment display connect to another programmable device, a CPLD (Complex Programmable Logic Device), and the CPLD interfaces with the FPGA. This module maps the points acquired by the user to the 7-segment display and samples the onboard button presses.
- **PS/2 interface**: it interprets the PS/2 keyboard scancodes for enabling a regular keyboard gameplay.
- **Top-level module** (Game logic): it instantiates the modules listed above and implements the game logic itself.

In the following sections, we give a more detailed overview of these building blocks and elaborate on their roles in the game.

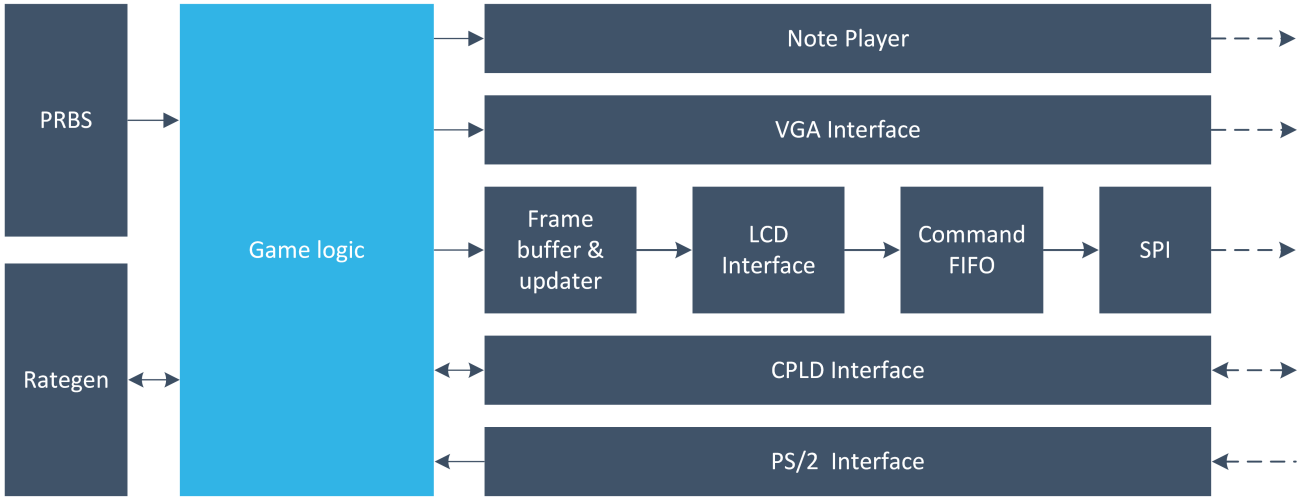


Figure 2: High-level connection diagram

2.1 SPI (Serial Peripheral Interface)

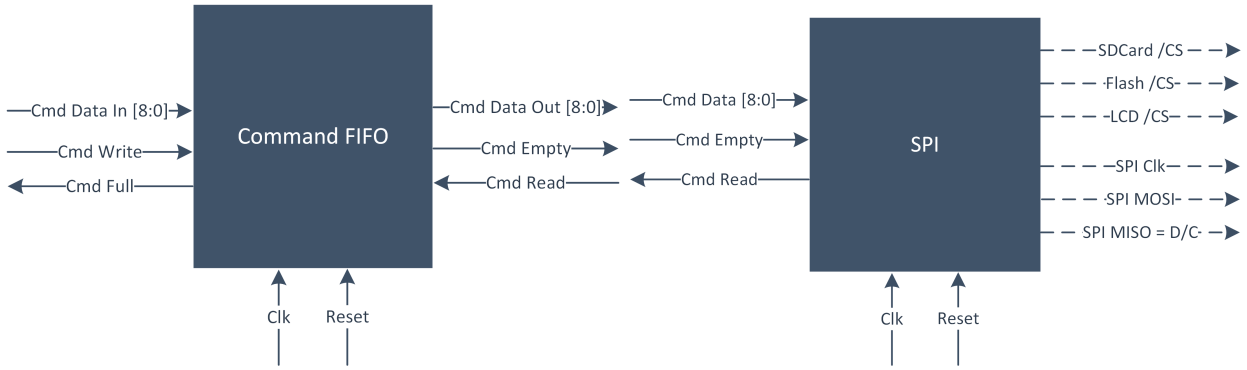
The SPI module implements the low-level communication medium with the LCD. The commands that will be sent are taken from the *Command FIFO*. If there's a command in the *Command FIFO*, we serialize it according to standard SPI.

The LCD (EA DOGS102-6) has a maximal input clock frequency of 33 MHz, so we generate a 25 MHz SPI clock from the 50 MHz input clock of the FPGA. Even though the data in the FIFO has a bit-width of 9-bits, only 8-bits serve as the actual data, 1-bit determines whether the data is a command or displayable content. This 1-bit sets the MISO line, which will be sampled by the LCD.

As there are more peripherals connected to the shared SPI bus, the chip select (CS) lines of these other devices are actively driven high (i.e. not selected).

2.2 Command FIFO

The commands for the LCD are stored in a 9-bit wide and 1024-entry deep FIFO, which is created via the Xilinx IP Core generator. The buffer uses dual-port block RAM, which also facilitates simultaneous write and read to this storage space.



(a) Command FIFO module (b) SPI module

Figure 3: Low-level modules towards the SPI LCD

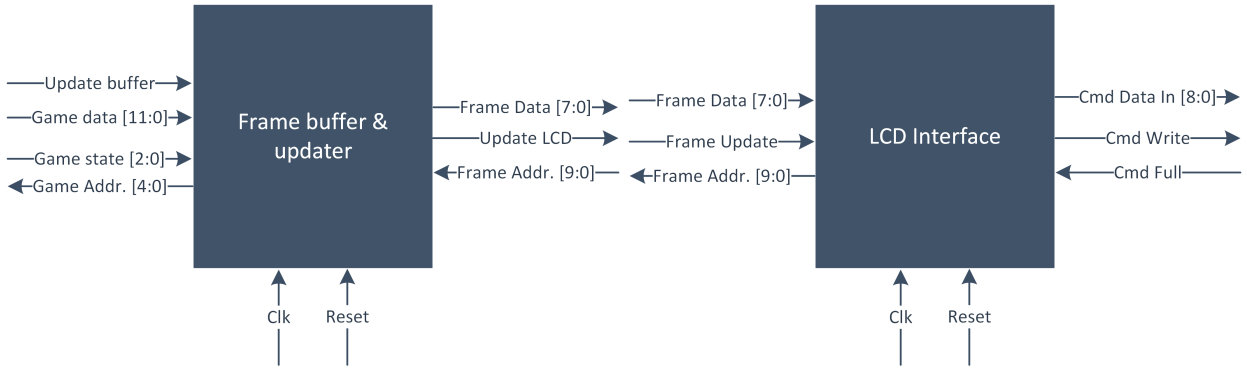
2.3 LCD interface

As both the *Command FIFO* and the *SPI* modules are general building blocks, we need a module that is specialized towards the LCD (EA DOGS102-6) present on the PCB. We need to initialize the display to make it able to interpret the displayable bytes correctly: we use a state machine that populates the *Command FIFO* with an initialization sequence, then we proceed with the bytes to be displayed.

We have to note that the display's driver only increments the column address, so at the beginning of each new row, we need to set the row and column addresses manually. The display's content is set according to the actual state of the *Frame buffer and updater*. To initiate a game space to LCD mapping, one needs to raise the module's *Update* input.

2.4 Frame buffer and updater

The purpose of this module is to map the game space/game logo (depending on the current state) to the SPI LCD's pixels. The present LCD has a display resolution of 102x64 pixels, which is organized into 8 pages and 102 columns. Each page has a height of 8 pixels, therefore we need to address 816 8-bit registers to fill the display.

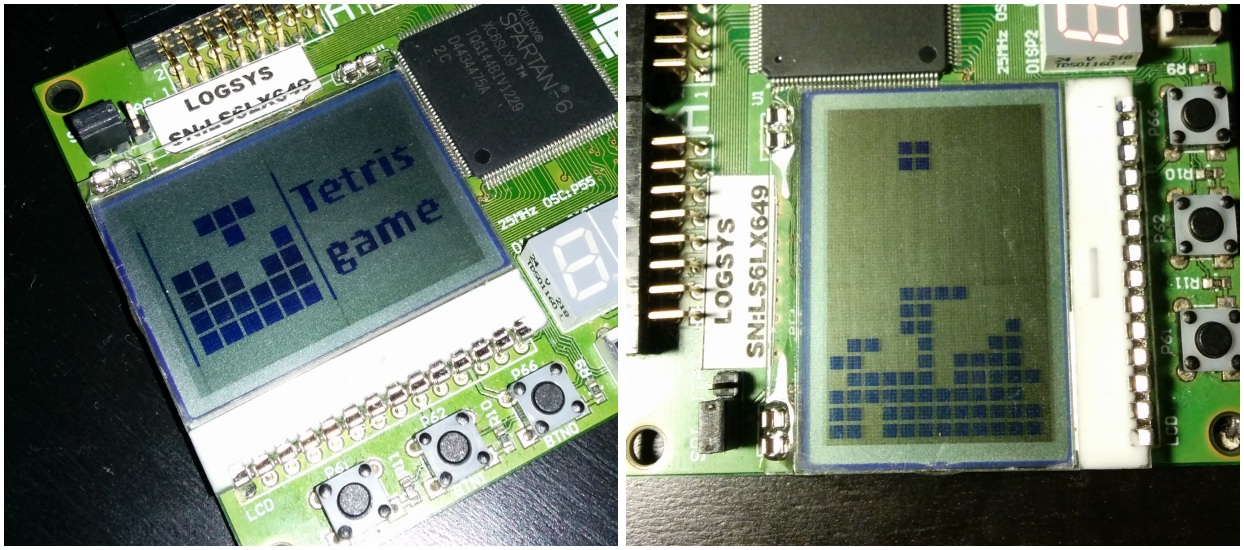


(a) Frame buffer and updater module (b) LCD interface module

Figure 4: High-level modules towards the SPI LCD

The game's welcome logo is a 102x64 pixel bitmap, in which "1"-bits encode black and "0"-bits encode transparent (white) pixels. This bitmap initializes the registers that hold the actual content of the display.

The game space itself consists of 20 rows and 12 columns of square blocks, and this can be filled with the falling elements. Each square block in the game space is mapped to a group of 4x4 pixels on the SPI LCD.



(a) Initial game logo (b) Actual game play

Figure 5: SPI LCD examples

2.5 PRBS (Pseudo Random Binary Sequence) generator

The random falling blocks are selected using a PRBS (Pseudo Random Binary Sequence) generator. The module's 6-bit output is used to select one of the building blocks. The generator polynomial is

$$PRBS15 = x^{15} + x^{14} + 1 \tag{1}$$

The initialization sequence is *100101010000000*, with which the module shares the same properties as the PRBS module present in DVB-T transmitters.

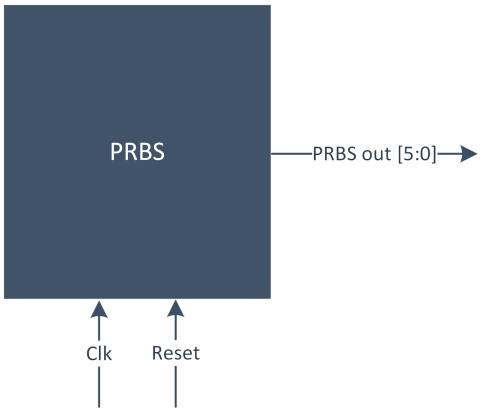


Figure 6: PRBS module

2.6 Rate generator

This module manages the speed of the falling blocks, and additionally, it also determines the rate of the game melody. By default, every building block is falling with a speed of 1 $\frac{row}{second}$. However, this default falling speed may be altered by selecting a different game difficulty.

In case the user positioned a building block to his/her liking and wants to drop that, by pressing the down arrow, the block's descent rate increases to 10 $\frac{row}{second}$.

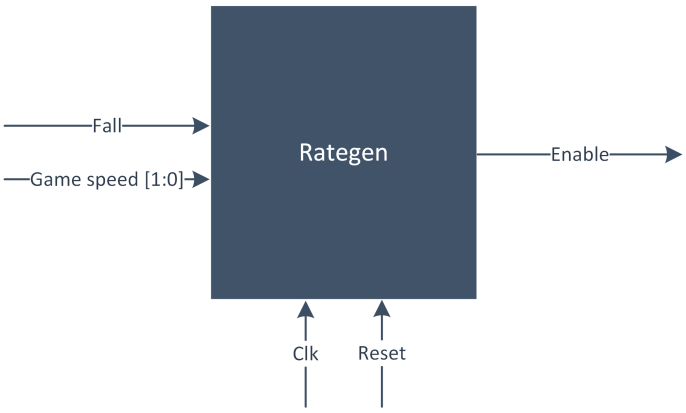


Figure 7: Rate generator module

2.7 CPLD interface

This hardware logic interfaces with yet another programmable device on the PCB, namely a CPLD. The detailed introduction of this other programmable device would be out of the scope of this high-level overview, therefore, for more information, refer to [1].

The module's responsibility is to read in the state of the buttons present on the printed circuit board (PCB) and transfer the number of eliminated rows to the 7-segment display. It merely acts as a bi-directional shift register: it outputs data on the generated clock's falling edge and samples its input on the rising edge. The generated clock for the CPLD is approximately 24 kHz. Lastly, this module also transforms the continuous button presses to single pulses, so that only pulses are forwarded to the rest of the hardware logic.

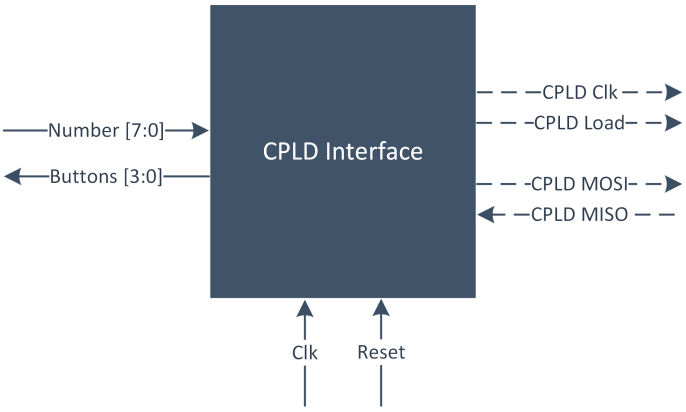


Figure 8: CPLD interface module

2.8 PS/2 interface

The PS/2 interface enables connectivity with a standard PC keyboard. Only a handful of scancodes [2] are interpreted and converted to pulse signals, that are propagated within the rest of the logic. To register a single keystroke, not only does the key need to be pressed, but it also needs to be released. This mitigates the problem of a keypress being registered multiple times. We summarized the interpreted button presses in the table below.

Physical button	**Scancode upon press**	**Scancode upon release**
Enter	0x5A	0xF0 0x5A
Left	0xE0 0x6B	0xE0 0xF0 0x6B
Right	0xE0 0x74	0xE0 0xF0 0x74
Up	0xE0 0x75	0xE0 0xF0 0x75
Down	0xE0 0x72	0xE0 0xF0 0x72
Space	0x29	0xF0 0x29
Esc	0x76	0xF0 0x76
M	0x3A	0xF0 0x3A
P	0x4D	0xF0 0x4D

Table 1: Registered button presses from the PS/2 keyboard

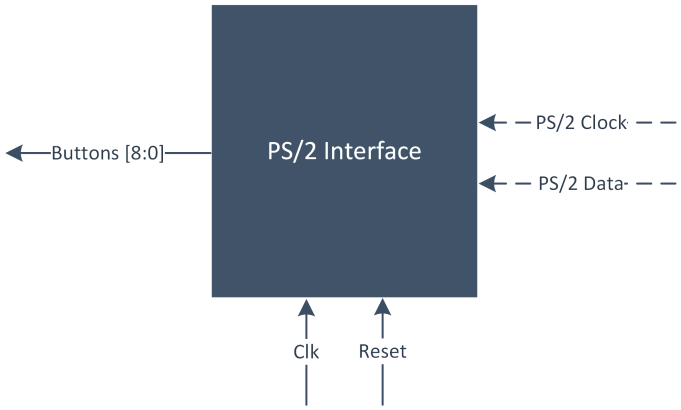


Figure 9: PS/2 interface module

2.9 Note player

This module supplies the melody of the game with the available piezoelectric buzzer. As the design only relies on the FPGA's internal resources (i.e. no SD card data is used or auxiliary content in the external flash storage), fitting an MP3 or PCM file into block RAM would have been infeasible due to the used FPGA's limited resources.

Therefore, the melody is stored as sheet music [4] in block RAM, and the musical frequencies are recreated via direct digital synthesis (DDS). DDS is a method of creating arbitrary frequencies (up to a certain precision) with a given sampling frequency and a phase register.

The idea of DDS is simple: we have an N-bit phase register, and we increment the phase register by one at each tick of the sampling frequency. The output of the signal is the most significant bit (MSB) of the phase register. Repeating this process long enough, our output will be a $\frac{f_s}{2^N}$ Hz square wave. We can increase the output signal's frequency by incrementing the phase register by a number greater than 1 (ΔP).

$$f_{output} = \frac{f_s \cdot \Delta P}{2^N} \text{ [Hz]} \tag{2}$$

$$\Delta P = \frac{f_{output} \cdot 2^N}{f_s} \text{ [1]} \tag{3}$$

For arbitrary f_{output}, the increment is simply the rounded result of ΔP. Our sampling frequency is generated from the main clock source of the FPGA (50 MHz) divided by 2048, which results in a sampling frequency of 24414.0625 Hz.

Armed with this knowledge, in the first half of the block RAM we store the musical note indices and their length (whole note, half note, quarter note, break, or end) in the format: ((length << 4) | note). The note itself is used as a pointer for the 2nd half of the block RAM where we store the phase register increments. The length signals how long we need to hold that particular frequency before moving to the next note.

#	Note	Frequency	Phase register increment (ΔP)
0	C0	261.6 Hz	702
1	C#0	277.2 Hz	744
2	D0	293.7 Hz	788
3	E0	329.6 Hz	885
4	F0	349.2 Hz	937
5	G0	392.0 Hz	1052
6	A0	440.0 Hz	1181
7	B0	466.2 Hz	1251
8	C1	523.3 Hz	1405
9	C#1	554.4 Hz	1488
10	D1	587.3 Hz	1577

Table 2: Notes played through DDS

The table above contains the phase register increments for a 16-bit phase register (that is $N = 16$).

To maintain our and the neighbors' sanity, the melody generation can be turned off by pressing the M button on the PS-2 keyboard.

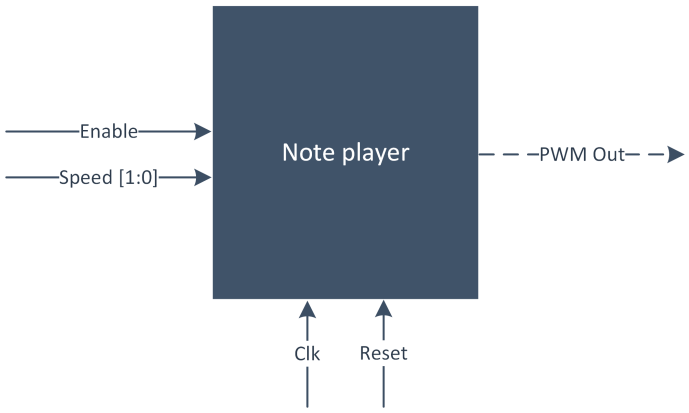


Figure 10: Note player module

2.10 VGA interface

Besides the SPI LCD output, the FPGA prepares a standard VGA output as well in a fixed resolution: 800x600 pixel, 72 Hz refresh rate. Not only do we display the play area, but also the player's actual score, the number of lines eliminated, and the current difficulty selected. Given the game has not started yet, we display some description of the hot-keys available. Once the game has started, the description goes hidden, and the next falling element will be displayed in its place.

The hardware, unfortunately, stipules some restrictions on the reproducible colors because we only have 2 bits per color channel, therefore total 256 colors. It is quite a step back from the 16 million colors (8-bits per color channel) that we're used to nowadays.

Figure 11: Example of a game text using 8-bits (left), and 2-bits (right) per color channel

The characters and the texts on the display are pre-rendered images and stored in the FPGA's block RAMs. The characters were rendered taking into account the display's color deficiency, so they were already rendered using four quantization levels (i.e. 2-bits). An example of using 2-bits instead of 8-bits is shown in Figure 11.

The individual elements on the screen are created by multiple submodules, whose output is multiplexed within the *VGA Interface* depending on the current pixel position.

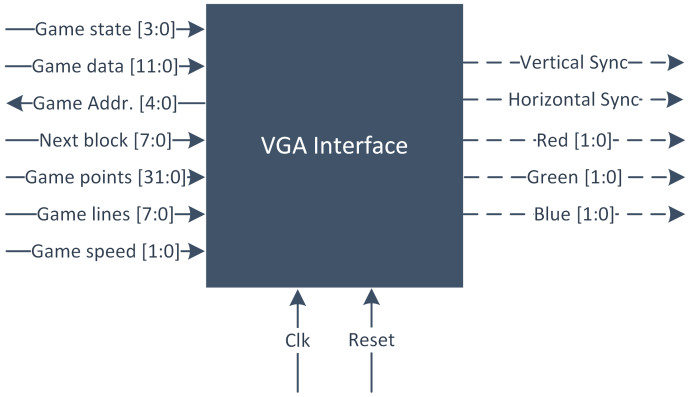


Figure 12: VGA interface module

2.11 Top-level module

The top-level module contains the game logic and all the previously introduced functional blocks.

2.11.1 Game space

The game space's size was determined by the SPI LCD's properties: squares that are still visible have a size of at least 4x4 pixels. One can place 20 rows and 12 columns of these squares onto the display to fully cover that, therefore totaling 240 square elements. Note that each Tetris building block uses precisely 4 of these squares in a different composition.

In the FPGA, the 20x12 game space becomes a 20x12-bit register array. Every new element that is *falling* is added to another 4x12-bit register array (sandbox), in which we can freely perform rotations and horizontal shifts. The height was determined such that all possible building blocks can fit in any orientation.

We maintain the currently falling element's position with respect to the top-left corner (with a horizontal and a vertical offset). The collisions are monitored via simple 'AND' relations between the game space and the sandbox.

The displayed game space is consequently the combination of the already landed elements in the 20x12 register array and the still falling element in the 4x12 sandbox. This is shown in Figure 13. The modules which are responsible for fetching the current game state (Frame buffer and VGA interface) use addressing to fetch a particular row of the combined game space.

2.11.2 Building blocks

The game contains seven building blocks that we can rotate and shift before they would land on the already settled game space. Each building block is stored on a single byte, as all the compositions can be described using only two rows and four columns.

2.11.3 Rotation

The rotation of the falling building block happens in a 4x4 matrix. The size of the matrix was chosen to fit even the largest element (1x4) in any orientation. We always fit the element to the top left corner of this matrix before the rotation begins. Once the rotation has been performed, we shift the element back to the top left corner of this 4x4 matrix to avoid the elements shifting sideways. In case there isn't enough space to perform the rotation due to the game space's already landed blocks, the rotation does not happen.

An example of the rotation is shown in Figure 15.

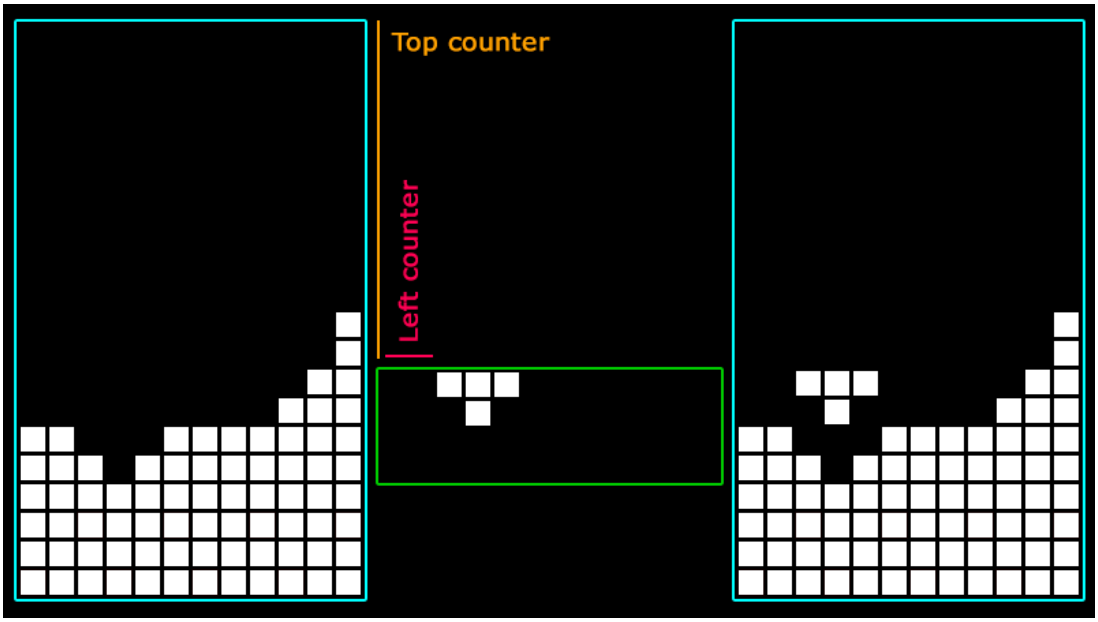


Figure 13: Hypothetical combination (right) of the landed elements (left) and the falling element (center)

(a) 0x0f (b) 0xe2 (c) 0xe8 (d) 0xcc (e) 0x6c (f) 0xe4 (g) 0xc6

Figure 14: Available Tetris building blocks with their binary representations

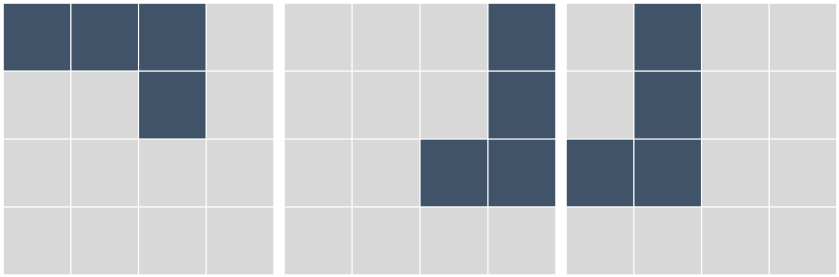


Figure 15: Rotating a building block in the 4x4 matrix, then applying the shift to avoid lateral movements

2.11.4 Course of the FPGA game

The FPGA implementation of the game follows suit with any regular Tetris game: the game starts by pressing the navigation button on the PCB, or the Enter key on the PS/2 keyboard. The previous game space gets cleared, and the new game space is displayed on both display modalities. According to the PRBS module's output, one of the seven building blocks is selected and it starts falling. This new building block is moveable as long as it does not collide with an already landed building block or hits the bottom of the game space. When the building block lands, a new one starts falling. Before the new building block commences its descent, we evaluate the present game space and eliminate the filled rows. Rows that have been filled are removed, and rows above them are shifted down. In case the new building block immediately hits an already landed block in the game space, the game ends.

Have Fun!

References

[1] Logsys Spartan-6 FPGA card, June 2015. http://logsys.mit.bme.hu/sites/default/files/page/2009/09/LOGSYS_SP6_FPGA_Board.pdf.

[2] Scancodes on Wikipedia, June 2015. https://en.wikipedia.org/wiki/Scancode.

[3] Tetris on Wikipedia, June 2015. https://en.wikipedia.org/wiki/Tetris.

[4] Tetris theme sheet music, June 2015. https://www.youtube.com/watch?v=IqagLvxYsGk.